Report Number: WUCS-85-02

1985-01-01

Specifying Software/Hardware Interactions in Distributed Systems

Gruia-Catalin Roman

This paper describes a system level specification approach that enables the designer to formulate and answer questions regarding the system's logical correctness and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc., must be considered. In the simplest terms, our concern extends beyond the traditional software correctness questions by addressing the issue of employing logical verification techniques to determine software correctness and performance characteristics when running on a particular distributed hardware architectures and using a particular operating system. A language called... **Read complete abstract on page 2.**

Specifying Software/Hardware Interactions in Distributed Systems

Gruia-Catalin Roman

Complete Abstract:

This paper describes a system level specification approach that enables the designer to formulate and answer questions regarding the system's logical correctness and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc., must be considered. In the simplest terms, our concern extends beyond the traditional software correctness questions by addressing the issue of employing logical verification techniques to determine software correctness and performance characteristics when running on a particular distributed hardware architectures and using a particular operating system. A language called CSPS (an extension of Hoare's CSP) is used in the illustration of the approach. Employing CSP as a base allows modelled systems to be verified using techniques already developed for verifying CSP programs.

Specifying Software/Hardware Interactions in Distributed Systems

Gruia-Catalin Roman

WUCS-85-02

March 1987

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

As appeared in the Proceedings of the 9th International Conference on Software Engineering, March 1987, pp. 126-139.

SPECIFYING SOFTWARE/HARDWARE INTERACTIONS IN DISTRIBUTED SYSTEMS

Gruia-Catalin Roman

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

ABSTRACT

This paper describes a system level specification approach that enables the designer to formulate and answer questions regarding the system's logical correctness and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc., must be considered. In the simplest terms, our concern extends beyond the traditional software correctness questions by addressing the issue of employing logical verification techniques to determine software correctness and performance characteristics when running on a particular distributed hardware architecture and using a particular operating system. A language called CSPS (an extension of Hoare's CSP) is used in the illustration of the approach. Employing CSP as a base allows modelled systems to be verified using techniques already developed for verifying CSP programs.

1. INTRODUCTION

We view a distributed system as a collection of application software modules, allocated over hardware components. This allocation, generally the responsibility of some operating system, may be static or dynamic: in a dynamic allocation, the mapping between software and hardware changes with time; in a static allocation, this mapping is constant. In this view, the system behavior is determined primarily by the software. Parts of the system other than software can affect its performance, however. For example, faults in the hardware may limit system capabilities, or the interaction of software and hardware in a particular allocation may degrade system performance. The workload, which is determined by the environment with which the system interacts, also affects system behavior.

Our research is aimed at developing system level models that enable the designer to formulate and answer questions regarding the system's logical correctness and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc., must be considered. In the simplest terms, our concern extends beyond the traditional software correctness questions by addressing the issue of *employing logical verification techniques to determine software correctness and performance characteristics when running on a particular distributed hardware architecture and using a particular operating system.*

The models are called *Virtual Systems* and represent either abstractions of existing systems or definitions of proposed systems. A virtual system consists of six components, each abstracting some aspect of a distributed system. The *Functionality* is an abstraction of the processes which carry out the system function (e.g., the applications software). The *Architecture* captures the overall hardware organization and distribution of the system. The *Scheduler* and the *Allocation* define the relationship between functionality and architecture and the changes which the relationship undergoes with time. This concept is an abstraction that encompasses many of the responsibilities one generally associates with an operating system, e.g., static and dynamic allocation of functions (in the functionality) to processors (in the architecture) as well as the allocation of time and space on an individual processor to the functions associated with it. The *Performance Specification* is an abstraction of both measurement probes and workload characteristics (the environment model is an integral part of the virtual system). The performance specification may be used to explicitly state the assumptions made by designers regarding the characteristics of the environment and of the system components to be utilized in the realization of the system. The performance specification is coordinated with the

rest of the model via the *Instrumentation.*

We have been experimenting with specifying virtual systems using a language called CSPS *(Communicating Sequential Processes with Synchronization).* An extension of Hoare's CSP [1], CSPS allows synchronization between multiple processes in addition to the I/O primitives of CSP. The functionality, architecture, scheduler, and performance specification are defined as closed communities of *communicating concurrent processes* while the allocation and instrumentation are defined as sets of *event synchronization rules.* The allocation, for instance, captures the interaction between software modules, hardware components, and scheduling policies by specifying synchronization rules among events in the functionality, scheduler and architecture. The synchronization rules establish a mapping between actions at one level of abstraction (e.g., software) and sequences of actions at a lower level (e.g., architecture). The result is a composite model that encompasses levels of abstraction which, initially, were described independently of each other. In this manner the effect of the architecture and scheduler on the software execution is factored into the proofs about the system's functionality.

To illustrate the event mapping idea let us consider an APL program as a realization of functionality and an APL machine (hardware interpreter) as a realization of architecture. Both functionality and architecture have states: for the APL program, the state depends on the value of local data and and the line number of the statement being interpreted; for the interpreter, the state depends on the program being interpreted, the interpreter's local data, and the program counter of the APL machine. In this common situation, the states of the program are mapped onto the states of the machine so that for every state of the program there is a corresponding distinct state of the machine. There may be unmapped intervening states in the machine, but not in the program. Each state transition in the program requires a sequence of state transitions in the machine until the states match again. The only differences between this illustration and the mapping defined by allocation are the presence of unmapped states in the functionality and the disregard for any events not involved in the synchronization rules.

We view CSPS only as an experimental notation scheme that allows one to exercise the modelling concepts prior to the development of a distributed system design language. Employing CSP as a base allows modelled systems to be verified using techniques already developed for verifying CSP programs [2,3,4] and, hopefully, will lead to the emergence of a uniform incremental strategy for verifying both logical and performance properties of distributed systems. Consequently, work on performance evaluation, resource allocation, and verification of concurrent processes may be drawn together by reducing some problems from the first two areas to equivalent problems in the third.

The remainder of this paper starts with a review of CSP which is immediately followed by an overview of the CSPS notation. Next, the structure of the virtual system and the motivation for that structure are explained. A simple ongoing example is used to illustrate the different components of a virtual system. The example is followed by an outline of the incremental proof strategy used to verify conformance to various types of functional (e.g., logical correctness) and non-functional (e.g., fault tolerance) system requirements. A discussion of the history of this effort and the pragmatics involved in modelling realistic systems concludes the paper.

2. CSP REVIEW

A full description of CSP is available in Hoare's original paper [1]. This section provides an informal definition of the syntax and semantics of the language. Three issues are addressed here: how to define a sequential process, how to specify concurrent execution of two or more processes, and how to describe communication between concurrent processes.

Each process definition takes the form

P :: S

where P is the process name and S stands for a composite statement. Ignoring communication, S is built out of three kinds of basic statements: assignment, guarded selection, and guarded iteration. The assignment operator is ":=" while guarded selection and iteration assume the following syntax, respectively,

[b1 → S1 # b2 → S2 # b3 → S3]

*[b1 → S1 # b2 → S2 # b3 → S3]

where b*i* is a boolean expression called a *guard* and S*i* is an arbitrary sequence of statements.

A guard is said to be *passable* if the boolean expression evaluates to *true;* otherwise it is said to *fail.* A guarded selection fails if all the guards fail. If one or more guards are passable, the non-deterministic selection of one of the passable guards is followed by the execution of the corresponding guarded statement. A guarded iteration is exited if all the guards fail. If one or more guards are passable, the non-deterministic selection of one of the passable guards is followed by the execution of the corresponding guarded statement and by another iteration.

Concurrent execution of a finite set of processes is specified using the statement

[P1 ‖ P2 ‖ ... ‖ Pn]

where no process may reference any variables subject to change in any other process.

Communication is accomplished via two I/O commands: *send* (e.g., P!x—meaning *send the value of x to process P*) and *receive* (e.g., Q?x—meaning *receive from process Q a value to be assigned to x*). I/O transfers take place only when one process names another as its destination for output and the second process names the first as its source for input. Such a situation is called a *matching* pair of I/O commands. If one process is ready to send to or receive from a process which is not yet ready to match the issued I/O command, the former must wait until both processes are ready.

One of the most important features of CSP is that in addition to the boolean expression a guard may also include an I/O command as in the example below:

[b1; P!x → S1 # b2; Q?y → S2 # b3 → S3]

When an I/O command is present, a guard is passable if the boolean expression is *true* and a matching I/O command is pending; the guard fails if the boolean expression evaluates to *false* or the process named in the I/O command is terminated. This last rule is referred to as the *distributed termination convention.*

3. CSPS

CSPS *(Communicating Sequential Processes with Synchronization),* as used in this paper, is a direct extension of CSP. The new feature, n-way synchronization, is motivated not by the desire to make a contribution to language theory but by practical considerations rooted in the modelling approach we have been pursuing. Every decision regarding notation is aimed at providing easy specification and modification of CSPS models. This section shows how n-way synchronization is specified in CSPS.

3.1. Labelled N-Way Synchronization

N-way event synchronization is defined as the simultaneous occurrence of N events. An *event* consists of an atomic action such as the execution of a simple statement, the selection of a guard, or the execution of an I/O command. The selection of a guard that includes an I/O command and the execution of the respective I/O command are seen as a single event.

In CSPS event synchronization is indicated through the use of *synchronization commands.* The simplest form of a synchronization command consists of a *synchronization label* and a *synchronization operator* (e.g., $):

P1:: ... **a** $; x:=0; ... **a** $ P!x; ...
P2:: ... *[x<6; **a** $ → x:=x+y] ... **a** $ **b** $; ...
P3:: ... *[x<6; **a** $ Q?y → x:=x+y] ...

where *a* and *b* are two synchronization labels.

A synchronization command may occur many times in the text of a single process. Any process whose text includes a particular synchronization command must always participate in the corresponding synchronization. A synchronization takes place only when each participating process is ready to execute the same synchronization command. If two or more synchronization and I/O commands appear together separated by blanks, in a *composite synchronization command,* all the commands must occur together—this feature enhances modularity of the models but adds no extra power to CSPS. CSPS extends the distributed termination convention to cover event synchronization commands. The last element in a guard may be an I/O command, a synchronization command, or a composite synchronization command. A guard fails if the boolean part is *false* or a terminated process is involved in any of the synchronizations or I/O commands appearing on the guard; a guard is passable if the boolean part is *true,* the I/O command may be executed, and all synchronizations may take place.

One might argue that two-way synchronization is easily described using I/O commands that pass dummy data and that probably n-way synchronization could be simulated by two way synchronization and, therefore, adds unnecessary complexity to the language implementation. Although we have been able to show that n-way synchronization may be simulated by employing I/O commands, it is our contention that even when two-way synchronization suffices, the use of synchronization commands offers the advantage of a cleaner separation of concerns, ease of understanding, and greater flexibility. Furthermore, for describing virtual systems, n-way synchronization is more intuitive and provides much needed economy of expression.

To illustrate the advantages of the notation we are proposing, let us consider a situation that might occur when modelling a real-time system consisting of some control software that drives a motor M. At some point in the system design one needs to model both the software and the motor. Because the design may undergo several iterations and because, once developed, the motor model should be available for use in future designs, the motor model should be independent of any

particular software with which it might be interfaced. A specification that satisfies these requirements may look as follows

```
M:: [ on:=false; off:=true;
      * [   on → run
        # on → on:=false; off:=true
        # off → idle
        # off → on:=true; off:=false]]
```

It provides a non-deterministic behavior description of the motor which must be interfaced with the control software logic whose task could be, for instance, to turn on and off the motor. Given the notation introduced earlier, the interfacing may be accomplished by simply adding two synchronization labels on the guards controlling the transitions between the running and idling states. (The distinction between the original model and the additions is clearly visible and reversible!)

```
M:: [ on:=false; off:=true;
      * [   on              → run
        # on; STOP $        → on:=false; off:=true
        # off               → idle
        # off; START $      → on:=true; off:=false]]
```

If at some later date it becomes necessary to control the rate (as discrete rotations let's say) a label STEP could be added on the first guard

```
on; STEP $ → run
```

The same label could be used as feed-back rather than control, i.e., to allow the software to count the number of rotations the motor executes, while the labels START and STOP may be used also to control a status light SL:

```
SL::   [ light:=false;
       *[   START $ → light:=true
          # STOP  $ → light:=false]]
```

The main motivation for n-way synchronization, however, rests with the need to match, i.e., synchronize, instantiations of the same event at three levels in the system: application software, operating system, and hardware. This will be illustrated fully in later sections of the paper.

3.2. Unlabelled Synchronization

Our search for notational convenience also led to the introduction of the "$$" as a synchronization command interpreted as: "*a process must resynchronize with every process with which it has synchronized since the last unlabelled synchronization or since the start of the process if no unlabelled synchronization occurred.*" (Note: More than one synchronization operator may be used if selective resynchronization is desired.)

The motivation for this type of synchronization rests with the fact that often what appears at the software level to be an atomic action may involve an entire sequence of events at the hardware level. The unlabelled synchronization provides a convenient way of signaling the end of the sequence of hardware level events without having to introduce additional labels:

```
software level action:
    STOP $; turn_motor_off; $$;
motor model actions:
    on; STOP $ → on:=false;
                 off:=true; $$;
```

By convention the software level action above may be also written in the following equivalent compact form

```
STOP $$ turn_motor_off;
```

3.3. N-Way Synchronization with Pattern Match

N-way synchronization, as introduced so far, is unable to accomplish the data transfers performed by I/O commands. To simulate

```
P:: ... Q!x ...
Q:: ... P?y ...
```

where x and y are integers, would require an infinity of labels, each encoding one possible value being transferred. This is an unacceptable limitation since the interdependencies between the software and the hardware may not be limited to control information but must consider the data being processed. The *n-way synchronization with pattern match* is a mechanism for accomplishing what may be seen as data transfer.

Here is an example of how to simulate an I/O exchange using synchronization with pattern match where x and y are assumed to be boolean:

Version 1 (special case, feasible when the domain of x and y is finite):

```
P:: ... PtoQ $ (x); ...
Q:: ... [   PtoQ $ (true) → y:=true
          # PtoQ $ (false) → y:=false] ...
```

Version 2 (general solution for simulating I/O exchanges):

```
P:: ... PtoQ $ (x); ...
Q:: ... PtoQ $ (y'); ...
```

PtoQ is the synchronization label which is used to determine (a priori) the set of processes that must

synchronize. The pattern definition appears as a list following the synchronization operator. For each variable in the pattern absence of a quote indicates that the *value* of the particular variable is part of the pattern for the respective synchronization; a single quote indicates that the value of the particular variable is part of the pattern for the respective synchronization but it is *indeterminate,* i.e., the variable will assume any value (within the restrictions of the variable type) that renders the pattern match successful. If a value is selected and the synchronization occurs the variable is actually assigned the particular value and the value may be used in the execution of following statements. The assignment of a value to an indeterminate variable does not affect the truth value of the boolean part of a guard.

A synchronization with pattern match is successful if a synchronization could occur in the absence of any patterns and the patterns associated with each instance of the label match. Two patterns match if there is at least one assignment of values to the indeterminate variables which would result in making the list of values identical. Given, for instance,

(1) K $ (x,y',z); where x=2, y=4 and z=4
(2) K $ (a',b,c); where a=8, b=7 and c=4
(3) K $ (u',v,w');where u=3, v=9 and w=1

(1) and (2) match leaving a=2 and y=7 but (1), (2) and (3) fail to match because of the inability to reconciliate b=7 with v=9. (When several synchronizations with pattern match are combined, the pattern evaluation proceeds from left to right. Since this feature is not used in the paper we will not discuss it any further.)

To illustrate a typical use of the synchronization with pattern match we show one way to provide the motor status to the control software:

```
M:: [ on:=false; off:=true;
    * [   STATUS $ (on)→ skip
      # on              → run
      # on; STOP $      → on:=false; off:=true
      # off             → idle
      # off; START $    → on:=true; off:=false]]
```

Although the synchronization with pattern match makes the use of I/O commands obsolete, we chose to retain both mechanisms in CSPS. CSP is used to describe models of various hardware, application software, and operating system components with the I/O commands specifying communication between components present in the same layer, e.g., application software. Interactions between the application software and the operating system and between the operating system and the hardware are described using synchronization commands. By maintaining this distinction the specification of a virtual system is easier to understand, configure (utilizing preexisting models of various components), modify, and analyze.

4. VIRTUAL SYSTEM

A *virtual system* captures the functionality, architecture, scheduling policies, and performance attributes of the system it models. This is accomplished by structuring the virtual system in terms of four communities of communicating processes that are related to each other via event synchronization. Communication within each community takes place via CSP I/O commands. The components of a virtual system are as follows:

Functionality is a model of the application software and consists of a community of communicating processes called *functions.*

Architecture is a model of the hardware organization and consists of a community of processes called *processors.*

Scheduler is a model of the runtime environment supporting the application software (i.e., operating system, language interpreter, etc.) and consists of a community of processes called *schedules.*

Allocation is a model of the behavioral interdependencies among the application software, hardware, and operating system making up a complete, potentially distributed, system. The allocation consists of *event synchronization rules* among events in the functionality, architecture, and scheduler. Each rule is identified by a unique synchronization label and is specified by the placement of synchronization commands bearing that label in the three communities of processes.

Performance specification is a formal definition of measures and assumptions (including behavior patterns) used by the designer in the analysis of a virtual system and consists of a community of (mostly non-communicating) processes called *actors.*

Instrumentation is a formal definition of the manner in which measures and assumptions are attached to the components of the virtual system. The instrumentation consists of synchronization rules between events in the functionality, architecture and scheduler, on one hand, and events in the performance specification, on the other.

The next four sections expand on the motivation for the structural components of the virtual system and provide simple illustrations based on a producer/consumer problem.

4.1. Functionality

The *functionality* represents the requirements for the application software. As such, it defines the interactions between the system and its operating environment as observed at the system user level. (This particular work makes no distinction between system functions and operating environment functions, but such a distinction could be made.) The software requirements are given by means of an operational model whose structural properties are relevant only when considered in relation to a particular architecture and scheduler, as shown two sections later.

Although the designer chooses to break down the functionality in one particular way to take best advantage of the available or postulated processors, there are many aspects of the functionality that may be evaluated by considering the functionality alone, e.g., logical correctness with respect to some externally stated criteria, freedom from behavior anomalies such as deadlock, and some primitive performance characteristics (e.g., relative frequency of execution of certain functions for some class of environmental inputs). Although these proofs assume no interference from the operating system and the hardware and the availability of adequate resources (e.g., storage space), they may be used later in proofs about the system as a whole.

To start the example let us consider a producer P which generates several values and passes them, one at a time, to a link L; the link L forwards each value to a consumer C; the consumer C receives a value from link L and computes the sum of all the values received so far. This functionality may be described as follows:

```
[ P || L || C ]

P:: [ x:=0; *[ x<3 → x:=x+1; L!x ]]

L:: *[ P?y → C!y ]

C:: [ u:=0; *[ L?z → u:=u+z ]]
```

The specification is written in CSP. It is self-contained and makes no reference to any operating system and hardware capabilities or resources.

4.2. Architecture

The *architecture* models the physical structure of the system and the behavior of the individual components that make it up. The components are not seen as physical devices but as resources required to support the application software. Associated with each model there is a *viewpoint* defining the level of abstraction and the purpose of the model. Different types of components are recognized at different levels of abstraction (e.g., nodes in a network, individual machines, large hardware components, etc.) and alternate models are used for each component depending upon the issue one chooses to address (the dynamics of storage management, communication behavior, fault detection, etc.).

The example for this section is communication via failing lines between two sites. An originator O selects a value to be sent to destination D and sends it on one of the unreliable interconnections I1 and I2:

```
[ O || I1 || I2 || D ]

O:: [ n:=0; up:=true;
      *[ up → n:=n+1; up:=false;
           *[   not(up); I1!n → up:=true
              # not(up); I2!n → up:=true]]]

I1:: [ up1:=true;  *[  up1; O?k1 → D!k1
                    # up1        → up1:=false]]

I2:: [ up2:=true;  *[  up2; O?k2 → D!k2
                    # up2        → up2:=false]]

D:: [ m:=0;        *[  I1?r → m:=m+1
                    # I2?r → m:=m+1]]
```

The message being transmitted is modelled by a message sequence number. The unreliable nature of the interconnections has been expressed as a non-deterministic choice in a guarded iteration. Selection of the first guard results in transmission of a value from O to D. Selection of the second guard results in the termination of the process modelling the respective interconnection.

In writing a processor model one must exercise great care to ensure the validity of the model. What may appear to be equivalent behaviors in one context may prove to be different in another. By using termination to model the interconnection failure, for instance, the model states that this particular failure may be detected by the other processors via the distributed termination convention, e.g., O could find out about a failure in I1 by using the statement

```
*[ I1!n → skip ]
```

which exits whenever I1 is terminated. An undetectable error could be modelled by entering an infinite loop:

```
I1:: [ up1:=true;  *[  up1; O?k1 → D!k1
                    # true       → up1:=false]]
```

Other methods are blocking the process by issuing an I/O command which will never be matched by the named process and failing the process by the use of an *abort* statement.

The model of the originator O illustrates another subtle implication of our architecture. The status of the interconnection is checked prior to selecting it. If this were not so, the model of the originator might take the form:

```
O:: [ n:=0; up:=true;
      *[ up → n:=n+1; up:=false;
            *[    not(up) → I1!n; up:=true
               # not(up) → I2!n; up:=true]]]
```

The change is that the I/O commands are guarded by *true,* rather than being part of the guards. This corresponds to selecting an interconnection for transmission without testing it to determine if it is functioning, and can result in deadlock: if the first guard is selected and I1 has failed, I1 will never ask for input from O and the originator will wait forever.

4.3. Scheduler and Allocation

The *scheduler* and the *allocation* describe the interdependency between functionality and architecture. The scheduler consists of a community of processes called schedules. Each schedule, together with the event synchronization rules that are part of the allocation, establishes a mapping between relevant events in one of the functions in the functionality and sequences of events involving one or more processors in the architecture. The allocation is said to be static whenever, for each function F, there is a unique schedule which always maps events in F to sequences of events in some unique processor P. The allocation is said to be dynamic whenever the events in F may be mapped into sequences of events involving more than one processor.

The nature of the scheduler depends upon the viewpoint adopted for the architecture and upon the characteristics of the function to processor mapping. This section illustrates three representative instances of this mapping: static one-to-one function/processor mapping, static many-to-one function/processor mapping, and dynamic one-to-one function/processor mapping.

4.3.1. Static allocation (one-to-one)

The static allocation of functions to processors does not require meaningful schedules if the processor models are very simple. The static allocation of the producer-consumer functionality to the origin-destination architecture with link L being allocated to interconnection I1 may be described by directly synchronizing events in the functionality with events in the architecture.

```
[ P ‖ L ‖ C ‖ O ‖ I1 ‖ D ]

P:: [ x:=0; *[ x<3 → x:=x+1; po $; L!x]]

L:: *[ li $ P?y → C!y ]

C:: [ u:=0; *[ L?z → cd $; u:=u+z]]

O:: [ n:=0; up:=true;
      *[ up; po $ → n:=n+1; up:=false;
            *[    not(up); I1!n→ up:=true
               # not(up); I2!n→ up:=true]]]

I1:: [ up1:=true;  *[  up1; li $ O?k1→ D!k1
                     # up1          → up1:=false]]

D:: [ m:=0;        *[  I1?r→ cd $; m:=m+1
                     # I2?r→ cd $; m:=m+1]]
```

The label po (may be read P on O) matches the production of a new x in P to the incrementing of the message counter n in O while the label cd matches the consumption of a new z in C to the incrementing of m in D. Similarly, L is allocated to I1 using the label li which matches a data passage through L to a data passage through I1. Because I2 is not used, it has been left out of the model, and in order not to have to change the descriptions for O and D, any reference to I2 is treated as a reference to a terminated process. Leaving I2 as an active component of the model could have caused undesirable behavior (erratic transmissions between D and O via I2).

4.3.2. Static allocation (many-to-one)

The sharing of a single processor by several functions may be illustrated by introducing two producers into the model used earlier.

```
[ P1 ‖ P2 ‖ L ‖ C ‖ O ‖ I1 ‖ D ‖ S ]

P1::   [ x:=0; *[ x<3 → x:=x+1; po1 $; L!x]]

P2::   [ x:=0; *[ x<2 → x:=x+1; po2 $; L!x]]

L:: *[    li $ P1?y → C!y
       # li $ P2?y → C!y]

C:: [ u:=0; *[ L?z → cd $; u:=u+z]]

O:: [ n:=0; up:=true;
      *[ up; po $ → n:=n+1; up:=false;
            *[    not(up); I1!n→ up:=true
               # not(up); I2!n→ up:=true]]]
```

```
I1:: [ up1:=true;   *[  up1; li $ O?k1 → D!k1
                     # up1            → up1:=false]]

D:: [ m:=0;         *[  I1?r→ cd $; m:=m+1
                     # I2?r→ cd $; m:=m+1]]

S:: *[    po1 $ po $ → skip
       # po2 $ po $ → skip]
```

The schedule S maps both P1 and P2 onto processor O but imposes no particular policy with regard to the use of this shared resource. S could be modified, however, to permit P1 and P2 to take turns.

```
S:: [ live1:=true; live2:=true;
      *[ live1 or live2 →
         live1:=false;
         *[ not(live1); po1 $ po $ → live1:=true]
         live2:=false;
         *[ not(live2); po2 $ po $ → live2:=true]]]
```

The complexity of the schedule is due to the need to avoid having one producer blocked by the termination of the other.

4.3.3. Dynamic allocation

The redefinition of the function/processor mapping may be embodied in the schedule's logic and is usually triggered by some local condition that develops in some part of the system. Our illustration for this section involves the static mapping of P and C to O and D, respectively, and the dynamic allocation of L to the interconnections I1 and I2. L is mapped to I1 until the failure of I1 causes L to be mapped to I2. This version of the example includes also the use of synchronization with pattern match and unlabelled synchronization.

The synchronization with pattern match is employed with the label po to make O transmit to D the same value that P sends to C in place of the message number used earlier. This is accomplished by having O use an indeterminate variable assignment (po $ (n');) which induces n in O to assume the same value as x in P (po $ (x);), i.e., the only value for which a match becomes possible. In the case of the label cd the pattern serves only as a check that the value z received by C is the same as the value r received by D.

The unlabelled synchronization is used to block L from accepting any more data until I1 (or I2) has completed the data transfer from O to D. Depending which interconnection is in use, $$ signifies a synchronization among L, S, and I1 or among L, S, and I2. If the interconnection I1 is up, the schedule S synchronizes only with I1 and L, first, via the labels li and li1 and, later, via $$. S continues to do so until I1 fails. Because the failure of I1 is modelled by termination, S can detect the failure (due to the distributed termination convention) and exits the first guarded iteration. From this point on, S accepts only synchronizations with L and I2 (via li, li1 and $$). The net effect is the dynamic reallocation of L from I1 to I2. It is important to note that L, I1, and I2 are not affected by the scheduling policy enforced by S.

```
[ P ‖ L ‖ C ‖ O ‖ I1 ‖ I2 ‖ D ‖ S ]

P:: [ x:=0; *[ x<3 → x:=x+1; po $ (x); L!x]]

L:: *[ li $ P?y → C!y; $$ ]

C:: [ u:=0; *[ L?z → cd $ (z); u:=u+z]]

O:: [ up:=true;
      *[ up; po $ (n') →
            up:=false;
            *[    not(up); I1!n→ up:=true
               # not(up); I2!n→ up:=true]]]

I1:: [ up1:=true; *[   up1; li1 $ O?k1 → D!k1; $$
                    # up1              → up1:=false]]

I2:: [ up2:=true; *[   up2; li2 $ O?k2 → D!k2; $$
                    # up2              → up2:=false]]

D:: *[    I1?r → cd $ (r)
       # I2?r → cd $ (r)]

S:: [   *[ li $ li1 $ → $$]
        *[ li $ li2 $ → $$]]
```

This particular version of the example will be used later to illustrate the incremental proof strategy. For this reason, it is important to point out that when one ignores the n-way synchronizations, for the purpose of analyzing a community of processes outside the context of the particular virtual system, any occurrence of an indeterminate variable (e.g., n') is treated as a random assignment of some value of the proper type.

4.4. Performance Specification and Instrumentation

The *performance specification* consists of a community of processes called actors whose interactions with the other components of a virtual system, defined by the *instrumentation,* take two distinct forms:

(1) they embody assumptions made by the designer about the characteristics of existing or envisioned system components—this is accomplished by stating the rules by which

various relevant performance attributes are being computed;

(2) they control non-determinism in the functionality, architecture, and scheduler by inducing patterns of behavior having particular characteristics.

When actors are used in a measurement capacity, they play a similar role to the reporting components of simulation languages. Events in the functionality, architecture, scheduler, and even performance specification may trigger predefined actions in the information recording actors. Take, for instance, the computation of the flows through I1 and I2. An actor A may take advantage of the synchronization labels li1 and li2 to determine which interconnection is used and to store this information into some internal counters f1 and f2.

```
A:: [ f1:=0; f2:=0;
      *[    li1 $ → f1:=f1+1
         # li2 $ → f2:=f2+1]]
```

If the flows are available, other data such as the average delay associated with the transmission via L may be deduced.

Fundamental to obtaining correct results is the need to ensure that a recording actor does not actually interfere with the behavior of the processes with which it is synchronized. Proving this for A is trivial: A is always ready to synchronize on li1 and li2 without imposing any restrictions in their ordering. B is provided below as a counterexample.

```
B:: [ f1:=0; f2:=0;
      *[ true →
           li1 $; f1:=f1+1;
           li2 $; f2:=f2+1]]
```

This actor forces alternate use of I1 and I2—the result is deadlock after the first use of the interconnection I1 by the link L. A sufficient condition for non-interference is to assure that the synchronized statements in a recording actor can be executed in arbitrary order, e.g., A permits the behavior {<li1>, <li2>}* while B is limited to the behavior {<li1,li2>}*. (Note: <a1,a2,a3> denotes a sequence, {<a>,<b>,<c>} denotes a set of sequences and {<a>,<b>}* denotes the set of all sequences containing only instances of <a> and <b>.)

Non-determinism may enter the model because of modelling a non-deterministic activity in the system or its environment (e.g., arrival of system requests) or a deterministic process whose original deterministic nature has been lost in the process of abstraction into the model. In many cases, something is known about the statistical nature of the non-determinism present. Actors may be defined so as to have a probabilistic behavior which, through instrumentation, may be imposed on functions, processors, and schedules.

To illustrate the way in which actors may impose a particular behavior pattern on other components of the system, let us consider an actor W whose role is to induce the failures of I1 and I2 after a fixed number of transmissions on each. (A probabilistic version of W could be defined but, for the sake of brevity, we chose to present a deterministic version of W.)

```
I1:: [ up1:=true; *[    up1; li1 $ O?k1 → D!k1
                     # up1; fi1 $       → up1:=false]]

I2:: [ up2:=true; *[    up2; li2 $ O?k2 → D!k2
                     # up2; fi2 $       → up2:=false]]
```

```
W::[ f1:=1; f2:=1;
     *[    f1<3; li1 $ → f1:=f1+1
        # f1=3; fi1 $ → skip
        # f2<5; li2 $ → f2:=f2+1
        # f2=5; fi2 $ → skip]]
```

The new labels fi1 and fi2 are used to force I1 and I2 to terminate.

Many of the features available today in discrete event simulation languages such as SIMULA [5] (e.g., random distributions for event arrival and processing time, reporting features, etc.) could be easily introduced in CSPS. This suggests that event synchronization should be given serious consideration as a potential mechanism for integrating discrete event simulation in design specification languages. The analogy to actual instrumentation of system components has a certain intuitive appeal. The ability to integrate the performance and functional issues while still maintaining a strong separation of concerns is desirable and has been attempted already by others (e.g., SREM [6]).

5. VERIFICATION STRATEGY

Our ultimate goal is to establish the basis for a system design specification language which offers the designer not only expressive power but also a variety of convenient logical verification techniques able to address both correctness and performance considerations in a uniform manner. Because of the size and complexity of real systems these techniques must be amenable to the development of incremental proofs and to recycling parts of existing proofs. Moreover, top-down system design makes it desirable to relate proofs corresponding to different levels of abstraction.

Our contributions toward achieving these ambitious goals have been limited so far to adapting CSP verification methods [2,4] for use with CSPS and to outlining a general strategy for the development of incremental proofs of system

properties. The remainder of this section illustrates our proof strategy on a slightly modified version of the example given under dynamic allocation. For the sake of brevity, the proofs are presented in an outline form. There are two parts to the illustration. The first one is concerned with the correctness of the software specification in isolation. Because n-way synchronization is not involved in this proof, it serves also as a review for the CSP verification method adapted for use in this section. A sketch of the correctness proof for the system as a whole follows.

Among existing techniques, we found Soudararajan's *communication traces* (called communication sequences in [4]) both easy to use and compatible with the overall objectives of our work. Each process has an associated trace whose purpose is to record, in order, all the I/O commands executed by the process (including the exchanged values). When considering only I/O commands, a trace consists of pairs ((e),(v)) where e is a label uniquely identifying the pair of processes involved in the I/O and the direction of the data transmission (e.g., P sending to L may be indicated by PL) and v is the value being transmitted. (Note: the definition given in [4] has been modified to accommodate synchronization commands.) To accomodate distributed termination, whenever a guarded iteration terminates the trace is augmented by a series of dummy communications with all the processes whose terminations cause the exit from the loop. These dummy communications are identified by a τ in the value field and appear in alphabetical order. Similarly, whenever a process terminates the trace is augmented by a series of dummy communications with all the processes which could be affected by the process' termination. These dummy communications are identified by a ω in the value field and appear in alphabetical order.

A partial correctness proof starts by proving appropriate properties about the individual processes in isolation; next, a rule of parallel composition[1] is used to prove properties of the community. All the axioms and rules of inference are fully described in [4] and will not be repeated here. Instead, we provide a proof outline for the functionality consisting of the processes P, L and C. We start by supplying pre- and post-assertions for each process.

[1] Parallel composition [4] involves (a) joining all the pre-conditions of the involved processes except for the predicates of the form "T=<>" and (b) joining all the post-conditions with the addition of a compatibility constraint.

```
    { Tp=<> }
P::    [ x:=0; *[ x<3 → x:=x+1; L!x]]
    { Tp=<((PL),(1)),((PL),(2)),
          ((PL),(3)),((PL),(ω))> }

    { Tl=<> }
L::    *[ P?y → C!y ]
    { Value(Tl|PL])=Value(Tl|LC]) and
      Type(Tl)∈{<(PL),(LC)>}* and
      Tail(Tl)=<((PL),(τ)),
                ((LC),(ω)),((PL),(ω))> }

    { Tc=<> }
C::    [ u:=0; *[ L?z → u:=u+z]]
    { u=SUM_OVER(Value(Tc|LC])) and
      Type(Tc)∈{<(LC)>}* and
      Tail(Tc)=<((LC),(τ)),((LC),(ω))> }
```

where

(1) Tp, Tl, and Tc are the traces for P, L and C, respectively;
(2) <> denotes the empty sequence;
(3) $T\alpha[\beta]$, called the projection of $T\alpha$ with respect to β, denotes a trace obtained from $T\alpha$ by eliminating every pair whose first element does not contain β and by replacing the remaining pairs with pairs containing only β as first element and the value matched to β as the second element;
(4) Value($T\alpha$) denotes a sequence obtained from $T\alpha$ by replacing every pair by its second element and by disregarding the values τ and ω;
(5) Type($T\alpha$) works the same way but keeps the first member of the pair and ignores the entries for which the values are τ and ω;
(6) Size($T\alpha$) gives the number of trace entries for which the values are not τ and ω;
(7) Tail($T\alpha$) returns the longest tail of the trace $T\alpha$ having only entries with values τ and ω;
(8) SUM_OVER(seq) is a function that totals all the values appearing in some sequence of integers.

Applying now the parallel composition rule, one may deduce

```
{ true }
   [ P ‖ L ‖ C ]
{  Tp=<((PL),(1)),((PL),(2)),
        ((PL),(3)),((PL),(ω))> and
   Value(Tl|PL])=Value(Tl|LC]) and
   Type(Tl)∈{<(PL),(LC)>}* and
   Tail(Tl)=<((PL),(τ)),
             ((LC),(ω)),((PL),(ω))> and
   u=SUM_OVER(Value(Tc|LC])) and
   Type(Tc)∈{<(LC)>}* and
   Tail(Tc)=<((LC),(τ)),((LC),(ω))> and
   Compatibility(Tp,Tl,Tc) }
```

But in this case the compatible[2] traces being

Tp=<((PL),(1)),((PL),(2)),
((PL),(3)),((PL),(ω))>
Tl=<((PL),(1)),((LC),(1)),((PL),(2)),((LC),(2)),
((PL),(3)),((LC),(3)),((PL),(τ)),
((LC),(ω)),((PL),(ω))>
Tc=<((LC),(1)),((LC),(2)),
((LC),(3)),((LC),(τ)),((LC),(ω))>

one can conclude

{ true } [P ‖ L ‖ C] { u=6 }

One more thing remains to be proven: termination. In our example this is rather easy to show: P terminates after three passes through the guarded iteration (due to strictly monotonic increase of the value of x); P's termination cascades to L and C (due to the distributed termination convention).

Next we need to consider the impact of the scheduler and the architecture—one would like to prove that the selected architecture and scheduling policies do not change the functionality of the system. The approach is the same. Properties of processors and schedules are proven in isolation and than the parallel composition rule is applied to all the processes making up the virtual system. Synchronization commands are treated in the same manner as the I/O commands except that (1) the value part of the trace is defined as the matched pattern, if a synchronization with pattern match is involved and as *nil* otherwise and (2) if several synchronization and I/O commands are involved, the pair entered in the trace contains as a first element a list of all the labels involved and as a second element a list of corresponding values (e.g., <((li1,OI1),(nil,1))>). As before, dummy communications and synchronizations are added at the exit from an iteration (e.g., <((li1,OI1),(τ,τ))>) and when a process terminates (e.g., <((li1,OI1),(ω,ω))>).

An outline of the proofs for the individual processes in the virtual system is given below. Please note, that for the sake of brevity complete assertions are provided only for the functions P, L, and C and for the schedule S. For the same reason unlabelled synchronizations have been eliminated. This triggered some code replication in L, the elimination of the label li and the introduction of the labels di1 and di2. Otherwise, the system as a whole is unchanged.

---

[2] In the simplest form, compatibility states that two communicating processes see the same sequence of actual communications. In addition, the distributed termination convention requires that all termination conditions are satisfied when an iteration exit takes place, i.e., every pair ((e),(τ)) in one process matches a ((e),(ω)) in the trace of the other process.

{ Tp=<> }
P:: [x:=0; *[x<3 → x:=x+1; po $ (x); L!x]]
{ Tp=<((po),(1)),((PL),(1)),((po),(2)),((PL),(2)),
((po),(3)),((PL),(3)),((PL),(ω)),((po),(ω))> }

{ Tl=<> }
L:: *[li1 $ P?y → C!y; di1 $
li2 $ P?y → C!y; di2 $]
{ Value(Tl[PL])=Value(Tl[LC]) and
Type(Tl)∈{<(li1,PL),(LC),(di1)>,
<(li2,PL),(LC),(di2)>}* and
Tail(Tl)=<((li1,PL),(τ,τ)),((li2,PL),(τ,τ)),
((di1),(ω)),((di2),(ω)),((LC),(ω)),
((li1),(ω)),((li2),(ω)),((PL),(ω))> }

{ Tc=<> }
C:: [u:=0; *[L?z → cd $ (z); u:=u+z]]
{ u=SUM_OVER(Value(Tc[LC])) and
Value(Tc[LC])=Value(Tc[cd]) and
Type(Tc)∈{<(LC),(cd)>}* and
Tail(Tc)=<((LC),(τ)),((cd),(ω)),((LC),(ω))> }

{ To=<> }
O:: [up:=true;
*[up; po $ (n') →
up:=false;
*[not(up); I1!n → up:=true
not(up); I2!n → up:=true]]]
{ (Value(To[po])=Value(To[OI1;OI2]) or ...) and ...}

{ Ti1=<> }
I1:: [up1:=true;
*[up1; li1 $ O?k1 → D!k1; di1 $
up1 → up1:=false]]
{ Value(Ti1[OI1])=Value(Ti1[I1D]) and ...}

{ Ti2=<> }
I2:: [up2:=true;
*[up2; li2 $ O?k2 → D!k2; di2 $
up2 → up2:=false]]
{ Value(Ti2[OI2])=Value(Ti2[I2D]) and ...}

{ Td=<> }
D:: *[I1?r → cd $ (r)
I2?r → cd $ (r)]
{ Value(Td[I1D;I2D])=Value(Td[cd]) and ...}

{ Ts=<> }
S:: [*[li1 $ → di1 $]
*[li2 $ → di2 $]]
{ Ts∈{<((li1),(nil)),((di1),(nil))>}* ×
{<(li1),(τ)>} ×
{<((li2),(nil)),((di2),(nil))>}* ×
{<(li2),(τ)>} ×
{<((di1),(ω)),((di2),(ω)),
(li1),(ω)),(li2),(ω))>} }

where

(1) × represents the pairwise concatenation of members of the sets on which it opearates;
(2) $T\alpha[\beta;\gamma]$ denotes a trace obtained from $T\alpha$ by eliminating every pair whose first element does not contain β or γ and by replacing the remaining pairs with pairs containing only β and γ (alone or together) in the first element and the values matched to β and γ as the second element.

The application of the parallel composition rule allows us to conclude the following

(1) u=6—which is what software correctness required;
(2) Value(To[OI1;OI2])=Value(Td[I1D;I2D])—which states that the sequence of values received by D must be the same as the one sent by O, regardless of which interconnection was used. The label cd ensures that if this condition is not met the system blocks and therefore it never terminates;
(3) Size(To[OI1;OI2])=Size(Td[I1D;I2D])=3—which states that three transmissions occurred via I1 and I2 before they terminated.

None of the conclusions we have drawn so far hold if the virtual system does not terminate—we outlined a partial correctness proof and we have not shown total correctness. Any attempt to prove termination, however, runs into difficulty due to the non-deterministic behavior of I1 and I2 which may terminate prematurely and thus block L. Under these circumstances it is important to reformulate the entire issue of termination by asking the questions:

(1) *What conditions must be met in order to assure termination?*
(2) *What condition prevents termination and where?*

These are the types of questions a designer must answer every time a new design proposal is put forth; successive refinements will eventually lead to a virtual system for which termination may indeed be proven.

We found all the techniques in use today to be of limited value in answering this kind of questions—they follow a paradigm where one must first determine all the circumstances under which blocking may be present and later prove that none of the situations can actually occur. Since the first step is where the designer is most likely to fail when analyzing a complex system, design errors may remain undetected. It appears to us, however, that the behavior information contained in the communication traces could be exploited to a greater extent in order to deal with some aspects of termination.

Considering our example again, one can argue that the normal communication behavior of all the processes may be abstracted as regular sets, if one ignores the values being transmitted:

$$
\begin{aligned}
Rp &\subseteq \{<po,PL,po,PL,po,PL>\}\\
Rl &\subseteq \{<(li1,PL),LC,di1>,\\
&\quad\ <(li2,PL),LC,di2>\}^*\\
Rc &\subseteq \{<LC,cd>\}^*\\
\\
Ro &\subseteq \{<po,OI1>,<po,OI2>\}^* \times \{<po>,<>\}\\
Ri1 &\subseteq \{<(li1,OI1),I1D,di1>\}^*\\
Ri2 &\subseteq \{<(li2,OI2),I2D,di2>\}^*\\
Rd &\subseteq \{<I1D,cd>,<I2D,cd>\}^*\\
\\
Rs &\subseteq \{<li1,di1>\}^* \times \{<li2,di2>\}^*
\end{aligned}
$$

where $R\alpha$ represents the regular set for process α.

By using the compatibility constraint one may develop the following necessary termination condition:

$$
\begin{aligned}
Rp &\subseteq \{<po,PL,po,PL,po,PL>\}\\
Rl &\subseteq \{<(li1,PL),LC,di1>\}^*n1 \times\\
&\quad\ \{<(li2,PL),LC,di2>\}^*n2\\
Rc &\subseteq \{<LC,cd>\}^*3\\
\\
Ro &\subseteq \{<po,OI1>\}^*n1 \times \{<po,OI2>\}^*n2\\
Ri1 &\subseteq \{<(li1,OI1),I1D,di1>\}^*n1\\
Ri2 &\subseteq \{<(li2,OI2),I2D,di2>\}^*n2\\
Rd &\subseteq \{<I1D,cd>\}^*n1 \times \{<I2D,cd>\}^*n2\\
\\
Rs &\subseteq \{<li1,di1>\}^*n1 \times \{<li2,di2>\}^*n2
\end{aligned}
$$

where $n1 \geq 0$, $n2 \geq 0$ and $n1+n2=3$.

From this it becomes apparent that premature termination of I1 and I2 will cause problems.

There are several reasons why conditions such as the one above are not necessary and sufficient:

(1) Local behavior patterns within a single process may prevent termination. This could happen in I1, for instance, if one uses *true* in place of up1 on the second guard

```
I1:: [ up1:=true;
     *[   up1; li1 $ O?k1 → D!k1; di1 $
        # true             → up1:=false]]
```

(2) The analysis ignored the pattern matches. With a different schedule, it is possible for D to receive values from O in the wrong order which would cause blocking between P and D on the synchronization cd. A more refined behavior analysis may be considered in such cases. In our example, by using the behavior of S we can show that the combined behavior of I1 and I2 may be described by

$$\{<(li1,OI1),I1D,di1>\}^*n1 \times \{<(li2,OI2),I2D,di2>\}^*n2$$

from which it is easy to see that no blocking takes place at the synchronization cd.

(3) Nondeterminism is known to lead to blocking situations which are not apparent when using regular sets.
(4) Regular sets, like most other behavior models, fail to capture the distributed termination convention.

Despite these technical difficulties, the notion of using behavior analysis for the purpose of simplifying the verification task has great intellectual and practical appeal.

The proof outline presented in this section is indicative of the potential that exists today to analyze distributed system designs by employing program verification methods and of the type of techniques that seem to be best suited to the task faced by the designer. The difficulty stems from the inherent complexity of the systems we try to describe, apparent even in the simple example used throughout the paper. Significant reductions in the complexity of the proofs may be brought about by breaking the proof into small parts that require a limited context to be built and understood. The strategy we are proposing involves three tracks:

(1) breaking the proofs into local and global issues on the line followed by Apt, et al., [2] and by Soundararajan [4] but considering two levels of locality, the process level and the virtual system component level, i.e., functionality, architecture, etc.;
(2) breaking proofs into levels of abstraction where a proof at one level is used to guide the lower level analysis as in the termination proof;
(3) reusing proofs as illustrated by the functionality proof appearing as a part of the system proof.

6. CONCLUDING REMARKS

The motivation for the work presented in this paper is rooted in the concept of a *Total System Design (TSD) Framework* [7]. The TSD Framework treats distributed systems as complex hardware/software aggregates whose design often requires careful consideration of the relationship between functionality and architecture in order to meet highly demanding constraints. The virtual system is a model that attempts to capture precisely the essence of this relationship.

The definition of the virtual system is motivated by traditional software engineering considerations:

(1) *Comprehensive coverage of key technical considerations.* Broad and integrated coverage of key technical considerations facing the distributed system designer is provided, including software and hardware organization, static and dynamic allocation, and performance modelling. Furthermore, in contrast to other distributed system design methodologies [8,9] where the architecture is treated only as a collection of computational sites, the virtual system attributes to processors arbitrary computational characteristics and behavior.
(2) *Separation of concerns.* The components of the virtual system correspond to well established areas of design expertise (e.g., software design, architecture design, resource allocation, performance modelling). The communities of communicating processes used to model the components capture design decisions that may be evaluated (up to a certain point) independently from the overall design or in a limited context. Dependencies among these components are expressed via synchronization rules that tie them together; e.g., functionality, architecture and scheduler (all having independent existence) are brought together by the allocation.
(3) *Complexity control.* The model permits independent elaboration, analysis, and modification of its components while allowing easy identification of the entities affected by particular changes. For instance, the architecture can be modified without changing the functionality, to determine how best to implement a required set of functions. Alternatively, the functionality can be restructured without changing the architecture, to determine how best to take advantage of a particular architecture or feature of the system hardware. In either case, the definition of the allocation may be used to determine the consequences of that change for the other components.

The first formal definition of the virtual system [10] was developed from a system theoretical perspective and lacked practicality, analyzability, and parsimony. The use of processes and event synchronization, first proposed in [11], has remedied the situation and brought us closer to what we envision a distributed system design language ought to provide. Most recent efforts have been directed primarily toward the development of a powerful, compact, unobtrusive, and easy to modify notation and the definition of its semantics. This paper illustrates some of the choices we have made but omits the more powerful aspects of the notation system which relay on an elaborate macro capability. The language work has been driven by a case study where a series of small but realistic virtual systems have been built [12]. By comparison, the producer/consumer illustration is not representative of the details of the modelling process.

7. REFERENCES

[1] Hoare, C. A. R., "Communicating Sequential Processes," *CACM* 21, No. 8, 1978, pp. 666-677.

[2] Apt, K. R., Francez, N., and DeRoever, W. P., "A Proof System for Communicating Sequential Processes," *ACM Trans. Prog. Lang. & Sys.* 2, No. 3, 1980, pp. 359-385.

[3] Levin, G. M. and Gries, D., "A Proof Technique for Communicating Sequential Processes," *Acta Informatica* 15, No. 3, 1981, pp. 281-302.

[4] Soundararajan, N., "Axiomatic Semantics of Communicating Sequential Processes," *ACM Trans. on Prog. Lang. and Sys.* 6, No. 4, 1984, pp. 647-662.

[5] Birtwistle, G. M. et. al., *Simula Begin,* New York: Petrocelli, 1973.

[6] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Trans. on Soft. Eng.* SE-3, No. 1, 1977, pp. 49-60.

[7] Roman, G.-C. et al. "A Total System Design Framework," *Computer* 17, No. 5, 1984, pp. 15-26.

[8] Estrin, G., "A Methodology for Design of Digital Systems," *1978 NCC Proc., 1978,* pp. 313-324.

[9] Mariani, P. M. and Palmer, D. F., *Distributed System Design,* IEEE Computer Society Press, 1979.

[10] Roman, G.-C. and Israel, R. K., "A Formal Treatment of Distributed Systems Design," *Requirements Engineering Environments,* Ohno, Y. (editor), OHM/North-Holland Pub. Co., 1982, pp. 3-12.

[11] Roman, G.-C. and Day, M. S., "Multifaceted Distributed Systems Specification Using Processes and Event Synchronization," *Proc. of 7'th Int'l Conf. on Soft. Eng.,* 1984, pp. 44-55.

[12] Roman, G.-C., Ehlers, M. E., Cunningham, H. C., and Lykins, R. H., "Toward Comprehensive Specification of Distributed Systems," Technical Report WUCS-86-8, Department of Computer Science, Washington University, Saint Louis, Missouri 63130, 1986.

Acknowledgement

The author is indebted to H. C. Cunningham, M. E. Ehlers, R. H. Lykins, and W. Chen for their review of this and previous versions of the paper. Their contributions to the development of the CSPS language and models are also acknowledged.